PCT

INTELLECTUAL PROPE
Internati nal Bt

INTERNATIONAL APPLICATION PUBLISHED UNDER

WO 9608948A2

(51) International Patent Classification :		(11) In
Not classified	A2	(43) International Publication Date: 28 March 1996 (28.03.96)

(21) International Application Number: PCT/IB95/00714

(22) International Filing Date: 30 August 1995 (30.08.95)

(30) Priority Data:
08/308,770 19 September 1994 (19.09.94) US

(71) Applicant: PHILIPS ELECTRONICS N.V. [NL/NL]; Groenewoudseweg 1, NL-5621 BA Eindhoven (NL).

(71) Applicant *(for SE only)*: PHILIPS NORDEN AB [SE/SE]; Kottbygatan 5, Kista, S-164 85 Stockholm (SE).

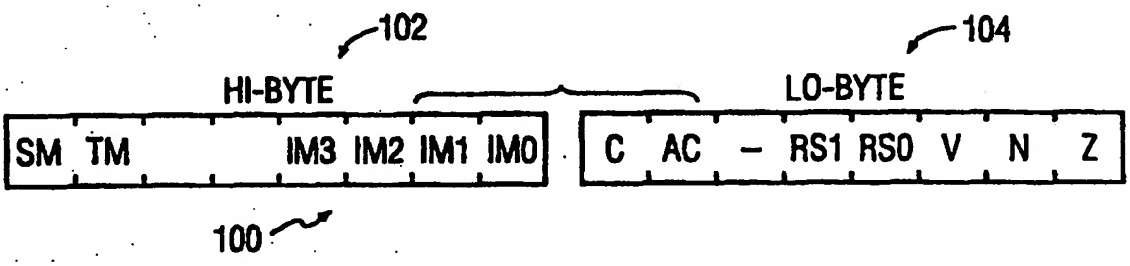
(72) Inventors: BIRNS, Neil; 741 Folsom Circle, Milpitas, CA 95035 (US). MIZRAHI-SHALOM, Ori; 2841 Burdick Way, San Jose, CA 95148 (US). GOODHUE, Gregory; 1345 Fairway Entrance Drive, San Jose, CA 95131 (US). RABELER, Thorwald; Stresemannallee 101, D-22529 Hamburg (DE).

(74) Agent: VERDONK, Peter, Lambert, Frans, Maria; Internationaal Octrooibureau B.V., P.O. Box 220, NL-5600 AE Eindhoven (NL).

(81) Designated States: JP, KR, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: METHOD AND APPARATUS FOR FAST MICROCONTROLLER CONTEXT SWITCHING



(57) Abstract

A microcontroller apparatus and method providing a program status word (PSW) including system status bits that can be stored and loaded along with a program counter (PC) during a context switch operation, such as when processing an interrupt or launching an application. The status bits indicate mode (system or user), designate which registers banks are used as general purpose registers and indicate the interrupt priority of the program being executed. By loading a PSW relevant to the task to be executed into the program status word register and loading a PC providing an entry point of the task into a fetch unit, the microcontroller immediately begins task execution. An interrupt vector of the microcontroller system includes the PSW and PC of the particular interrupt allowing immediate processing of the interrupt reducing system overhead tasks. A return from the task, such as via a return-from-interrupt, restores the PSW of the prior task as well as the PC from a system stack, thereby rapidly switching the context back to the prior task. By loading an application specific PSW and PC (and appropriate general purpose registers and special function registers is necessary), and then initiating a return-from-interrupt the microcontroller allows rapid initial task launching.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	**GB**	United Kingdom	**MR**	Mauritania
AU	Australia	**GE**	Georgia	**MW**	Malawi
BB	Barbados	**GN**	Guinea	**NE**	Niger
BE	Belgium	**GR**	Greece	**NL**	Netherlands
BF	Burkina Faso	**HU**	Hungary	**NO**	Norway
BG	Bulgaria	**IE**	Ireland	**NZ**	New Zealand
BJ	Benin	**IT**	Italy	**PL**	Poland
BR	Brazil	**JP**	Japan	**PT**	Portugal
BY	Belarus	**KE**	Kenya	**RO**	Romania
CA	Canada	**KG**	Kyrgystan	**RU**	Russian Federation
CF	Central African Republic	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CG	Congo			**SE**	Sweden
CH	Switzerland	**KR**	Republic of Korea	**SI**	Slovenia
CI	Côte d'Ivoire	**KZ**	Kazakhstan	**SK**	Slovakia
CM	Cameroon	**LI**	Liechtenstein	**SN**	Senegal
CN	China	**LK**	Sri Lanka	**TD**	Chad
CS	Czechoslovakia	**LU**	Luxembourg	**TG**	Togo
CZ	Czech Republic	**LV**	Latvia	**TJ**	Tajikistan
DE	Germany	**MC**	Monaco	**TT**	Trinidad and Tobago
DK	Denmark	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**US**	United States of America
FI	Finland	**ML**	Mali	**UZ**	Uzbekistan
FR	France	**MN**	Mongolia	**VN**	Viet Nam
GA	Gabon				

Method and apparatus for fast microcontroller context switching.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention is directed to a system for switching a microcontroller rapidly between tasks and, more particularly, to a system which uses program status word swapping to reconfigure a microcontroller when switching between separable tasks.

Description of the Related Art

Conventional microcontrollers which support multitasking typically provide a system or supervisor mode of operation under which system supervisor tasks are executed and a user or application mode under which various applications are executed. The system mode allocates access to particular segments of memory to each application and establishes constraints or parameters under which each application executes (for example, minimum priority level required to interrupt, right to access certain registers, etc.) The supervisor mode is also allowed to execute certain operations not allowed under the user mode (for example writing to a "protected" system configuration register). Each of the applications executing in the user mode can also require and be restricted from certain system resources and procedures. In typical microcontrollers the operation of switching between mode or applications involves a number of steps which require a considerable amount of time, sometimes called overhead, to execute. For example, when switching between first and second applications or tasks while in the user mode the contents of the general purpose registers of the first task must be saved and the contents of the general purpose registers of the second task must be loaded before the second task can be started. When interrupts occur the status of the system must be switched from user mode to supervisor mode which typically involves not only the storing and loading of registers as previously described but also the loading of the appropriate program status bits indicating system supervisor status, interrupt priority and other system status. As a result, a switch between user and supervisor modes has traditionally required a relatively large amount of time and the consequent overhead associated therewith. To switch between applications where the control is transferred through the supervisor mode is even more time consuming.

To improve performance of microcontrollers and reduce overhead during

switching between modes and applications, a technique is needed which rapidly switches between system and user mode, and between applications within the user mode, and to quickly establish key parameters for each mode which is being executed as well as for each routine or task to be executed.

When initially launching a user application, such as when a microcontroller is reset, the supervisor task in the conventional microcontroller typically also has to perform the operations necessary to allow a switch to the user mode as discussed above.

To improve performance during initial application launch a technique is needed for fast application start-up.

SUMMARY OF THE INVENTION

It is an object of the present invention to provide a microcontroller system with the ability to rapidly switch between system and user modes and between applications.

It is another object of the present invention to provide a program status word and associated microcontroller architecture that facilitates rapid context switching.

It is another object of the present invention to provide a microcontroller system that facilitates rapid application start-up or launch.

The above objects can be attained by a microcontroller system providing a program status word including system status bits that can be stored and loaded during any context switching operation, such as when processing an interrupt. The program status word indicates and controls microcontroller mode (system or user), selects which of several banks of registers are used as general purpose registers and controls interrupt priority. The system also associates a program counter with the program status word where the program counter indicates an entry point into the target task. By loading a program status word relevant to the task to be executed when the program counter is loaded with the location of the task entry point instruction, the microcontroller can immediately begin task execution. A return from the task, such as via a return from interrupt, restores the program status word of the prior task as well as the program counter from a stack, thereby rapidly switching the context back to the prior task. By loading the appropriate program status word and program counter (and appropriate general purpose registers if necessary), and then initiating a return from interrupt the microcontroller also allows rapid initial task launching or initiation.

These together with other objects and advantages which will be subsequently apparent, reside in the details of construction and operation as more fully hereinafter described and claimed, reference being had to the accompanying drawings forming a part

hereof, wherein like numerals refer to like parts throughout.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 depicts the hardware of the microcontroller of the present invention;

Figure 2 depicts a more detailed view of a part of the hardware of figure 1;

Figure 3 depicts the structure of the program status word 100;

Figures 4, 5 and 6 illustrate memory organization particularly the location of the general purpose registers;

Figures 7 and 8 depict the stacks of the microcontroller;

Figure 9 depicts a segment address;

Figure 10 illustrates an example of a vector table;

Figure 11 depicts a program counter and program status word on a stack during interrupt processing;

Figure 12 is a flow chart of the context switch operations of the present invention; and

Figure 13 depicts a program launch operation.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention provides a technique of rapidly switching between system and user modes of operation and of quickly establishing key parameters for each routine to be executed. The technique essentially involves loading the program status word (PSW) whenever any interrupt or other exception (trap, break instruction, etc.) occurs, with a value specified in program memory. Each specific interrupt source or exception is provided with a dedicated address in a vector table in a program memory (used for storing only instructions or code) where the PSW contents to be loaded for the corresponding service routine resides. The current PSW is pushed onto a stack before the new PSW loaded. Upon return from the interrupt, after the task is complete, the PSW contents are unloaded from the stack and again control program status. The PSW contains a bit which determines whether the machine is in system mode or user mode, register bank-select bits which determine which register bank is used by a routine, interrupt priority bits, and a bit to initiate trace mode, along with various system flags. When any interrupt, software or hardware trap, or break is detected, an instruction decoder latches the program-memory address where the new PSW contents are located, provides this address to an execution control unit, and initiates execution of a microcode program controlled branch-to-exception or call interrupt instruction ("CALLI").

During execution of this instruction, the execution-control unit pushes the current PSW contents in the PSW register onto the stack, retrieves the new PSW value from program memory, and loads this new value into the PSW register. Transfers to and from the PSW utilize an internal, 16-bit address/data bus and microcode controlled read/write control signals. When returning from an exception, (via a return from interrupt - "RETI" instruction) the execution control unit retrieves the previous PSW and PC contents from the stack and moves the PSW into the PSW register and loads the PC into the fetch unit, thus, returning execution to the prior interrupted task. The system mode is written into the PSW register from the PSW vector or stack during execution of the branch-to-exception and return-from-exception instructions. Pushes and pops during an interrupt/exception always use the system stack, and all bits of the PSW (including the system mode bit) are writable. The new PSW configuration takes effect immediately following the exception handling instruction.

When the microcontroller is in the system mode, to completely set up an environment for an application which needs to be run, and to launch the application, because of the mechanism described above and in more detail below, the system mode program can simply load the general purpose and special registers with the data necessary to start the application, load the system stack with the desired PSW contents and with the application's starting PC address, then execute an RETI instruction. Execution of this instruction will load the PSW from the stack, branch to the PC address popped from the stack, and begin execution. The PSW contents constructed by the system mode program prior to the launch will dictate which register bank the application will use, establish interrupt priority level, determine whether the program will run in system or user mode, and initialize any necessary flags.

The architecture of the microcontroller system 10 of the present invention is illustrated in figure 1. This system 10 includes a single chip microcontroller 12 that performs 16 bit arithmetic operations and includes internal instruction and data storage. The microcontroller 12 supports external devices 14 and 16 and, through 24 bit external address capability, supports sixteen megabytes of external instruction or program storage 18 and sixteen megabytes of external data storage 20. The microcontroller 12 includes a bus interface unit 22 which communicates with the external memories 18 and 20 over an external bi-directional address and data bus 24. The microcontroller 12 communicates with the external devices 14 and 16 through I/O ports 26 - 28 which are addressable as special function registers (SFR) 40. The ports 26-28 as well as other special function registers are addressable over an internal peripheral bus 42 through the bus interface unit 22. The data

memory 20 can also be accessed as off-chip memory mapped I/O through the I/O ports 26 - 28 which is access illustrated by the dashed line. The on-chip special function registers 40, some of which are bit addressable, also include the program status word (PSW) register 44 which is coupled to an interruption control unit 84 communicating with the external devices as well as the ALU 72, the execution unit 70 and decode unit 74 for flag and general control. The special function registers 40 also include an interrupt register 44. Timer registers 50 and a system configuration register (SCR) 54 containing system configuration bits are also found in the special function register space 40. The program status word register 44 is addressable over the peripheral bus 42 for general register operations and is also addressable over a connection to the internal bus 86 for other execution related operations, such as when loaded during a context switch. The bus interface unit 22 isolates the peripheral special function registers 40 from the microcontroller core 60. The core 60 includes a microcode programmable execution unit 70 which controls execution of instructions by an ALU 72 and the other units. The instructions decoded by a decode unit 74 are fetched from an internal EPROM 76, which is part of the instruction memory space, or from the external instruction memory 18 by a fetch unit 78. Static RAM 80, which is part of the data memory space, as well as general purpose registers of a register file 82 are also available for instruction and data storage.

During the rapid context switching of the present invention a staging register 90, as illustrated in figure 2 and as will be discussed in more detail later, is loaded by the decode unit 74 with a microcode address of the particular context switch instruction (for example, the interrupt instruction) to be executed and is also loaded with an interrupt vector address which is provided to fetch unit 78. The execution unit 70 operation during execution of the context switch instruction is controlled by microcode program instructions retrieved from a microcode ROM 92.

The program status word register 44 includes a program status word 100, as illustrated in figure 3. The PSW register 44 is a word register in the bit-addressable SFR space 40. The high byte 102 is a protected half containing the system/supervisor level flags. The second byte 104 contains all user level flags and functions as described below. The default PSW value, including the initial interrupt priority (which is set to maximum i.e all 1's until the system initialization code completes), is loaded upon a reset. This prevents spurious interrupt services before code initialization. Most arithmetic logic, and data transfer instructions update some or all of the status flags. PUSH and POP instructions, however, do not update any of the status flags. Update of the PSW status flags is also suppressed during

any write to the PSW. The data written to the PSW, such as during a load of the PSW in a context switch, takes precedence over normal flag updates. This applies to both bytes of the PSW during a byte write to one half of the PSW. C is the carry flag and the main function of this flag is to store the carry out of the most significant bit of an arithmetic operation. AC is auxiliary carry flag which is updated during arithmetic instructions with the carry out of the least significant nibble of the ALU. FO and F1 are user definable flags and may be read and written by user programs. These flags can be updated by a supervisor mode program when it is preparing to launch a user application. RS1 and RS2 are register bank select bits which identify one of the four groups or banks of registers R0 through R7 in the register file 82 which are active at a given time. The four register banks are also addressable directly and indirectly as the bottom 32 bytes of data memory. V is the overflow flag and is set by a twos complement arithmetic overflow condition during arithmetic instructions. P is the parity flag and this bit shows the even parity for the current contents of register R8. SM is the system mode bit flag. The system mode is selected at reset, can be changed during interrupt processing and is written by a return from interrupt (RETI) by popping the PSW. This ability is intended as an aid to multitasking applications as previously mentioned. TM is the trace-mode bit and is used to aid in the program development to allow instruction-by-instruction tracing. Z is a zero operation indication flag and after a data operation other than POP, PUSH, XCH, or SCHD, the Z flag is set to 1 if the operation returned a result of 0, otherwise the Z flag is cleared to 0. N is a negative operation indication flag and after a data operation other than POP, PUSH, XCH, or SCHD, the N flag is set to 1 if the operation returned a result with the sign bit (MSB) set, otherwise the N flag is cleared to 0. IM3 - IM0 are execution priority interrupt mask bits where these bits are used to identify the execution priority of the currently executing code. In the case of an interrupt, these bits will be set to the interrupt priority of the interrupt in progress. These bits can be also changed or written during interrupt processing allowing the interrupt handler or routine to change the interrupt priority of the interrupt handler after it has started executing. The microcontroller 12 supports saving and restoring of these bits during traps, interrupts, and return from interrupt, as well as providing lines from these bits through the interrupt control unit 84 to any interrupt control module outside of the core. The actual number of priorities implemented is variable, with a maximum of 16 levels. Writes to the IM bits are restricted to system mode code.

The data memory space 118 is segmented into 64K byte pages 120, accessed via indirect addressing modes (the first 1K block of each page is also directly addressable).

There are four banks of registers R0 through R7 (see figures 4, 5 and 6) starting at address 0 in the on-chip RAM (in the register file 82) and going up to address 1F hexadecimal. One of the four banks is selected as the active bank by the two bits R0 and R1 in the PSW 100. The selected bank appears as the general purpose registers. As previously mentioned the bank of registers can be changed during a context switch by loading a new PSW. Register R7 is the stack pointer, either the system stack pointer or the user stack pointer depending on whether the microcontroller 12 is in the system or user mode. Register R7 is used during push and pop operations during a context switch.

As previously stated, the microcontroller 12 supports a program memory 18 with an addressable space of 16 megabytes. Program memory target addresses referenced by jumps, calls, branches, traps and interrupts, under microcode program control, must be word aligned. However, the return address from subroutines or interrupt handlers can be on either odd or even boundaries. For instance, a branch instruction may occur at any code address, but it may only branch to an even address.

In the microcontroller 12 the stack 130, as illustrated in figures 7 and 8, grows downward from high to low addresses. The microcontroller 12 architecture supports a LIFO (last-in first-out) stack. At any given time, the stack pointer (SP), which is the contents of register R7, points to the last word pushed onto the stack. When new data is pushed, such as the PSW and PC during a context switch, the stack pointer is decremented prior to writing to memory. When data is popped from the stack, the stack pointer is incremented after the data is read from memory. Since the microcontroller 12 stores data in the memory MSB first, the stack pointer always points to the LSB of a word written on the stack. Stack operations are facilitated by two stack pointers a user stack pointer (USP) and a system stack pointer (SSP) located in the registers of register file 82. The 16-bit stack pointers are customary top-of-stack pointers, addressing the uppermost datum on a push-down stack. It is referenced implicitly by push and pop operations, subroutine calls, and interrupt operations. The stack is always word aligned. Any push to the stack (byte/word) decrements the stack pointer by two (SP = SP-2) and any POP (byte/word) increments the stack pointer by two (SP = SP+2). The stack alignment thus ensures that all stack operations are on word boundaries (even address), eliminating alignment issues and reducing the interrupt latency time as well as for other 16-bit or larger stack operations. In the microcontroller 12, a push or pop stack operation does not update any status flag in the PSW. Also, all status flag updates are suppressed during a write to the PSW register 44. The user stack pointer (USP) register may be written as well as read using its register address, and by doing so the user stack may be

placed anywhere in the segment where the stack resides. F r the system stack, this is always page 0 (16-bit address only), while the user stack space is identified by the data segment (DS) register. The stack may be as deep as the available memory on its memory page permits. The bottom limit of the stack in all pages is set to address 80 hex i.e a microcode program controlled stack overflow trap occurs at that address. However, there is an extra 64 byte space below this stack bottom limit, out of which 22 bytes could be reserved to accommodate the worst case scenario of having 16 bytes for a multiple register (RO-R15) PUSH and an additional 6 bytes to store the stack frame for the overflow trap. At power-on reset, the SP is always initialized to 100 hex i.e one byte above minimum on-chip RAM space (256 bytes). Since SP is pre-decremented prior to a PUSH, a word-aligned stack would grow from FE downwards.

The microcontroller provides a two-level user/supervisor protection mechanism. These are the user or application mode and the system or supervisor mode. In a multitasking environment, tasks in a supervisor level are protected from tasks in the application level. As noted previously, the microcontroller has two stack pointers (in the register file) called the system stack pointer (SSP) and the user stack pointer (USP). In multitasking systems one stack pointer is used for the supervisory system and another for the currently active task. This helps in the protection mechanism by providing isolation of system software from user applications. The two stack pointers also help to improve the performance of interrupts. If the user stack for a particular application would exceed the space available in the on-chip RAM 80, or on-chip RAM 80 is needed for other time critical purposes (since on-chip RAM 80 is accessed more quickly than off-chip memory 20), the user stack can be put in off-chip RAM 20 and the interrupt stack (using the system SP) may be put in on-chip RAM 80. The system stack is always forced to data memory segment 0 (the first 64K bytes of data memory), while the user stack is located on the segment chosen by the DS (Data Segment) register. The two stack pointers share the same register address. The stack pointer that will be used at any given time, and that will "appear" in the register file, is determined by the system mode bit (SM) in the program status word (PSW) register 44. The microcontroller 12 stack is automatically set via microcode program control to use the user stack pointer (USP) whenever code is executing in the user mode and the system stack pointer (SSP) when code is executing in the system mode. Shadowing the two SPs allows the same procedures to run in both system and user modes, as in a compiler run-time package. The microcontroller 12 begins operation after reset in the system mode using the system stack for pushes, pops, subroutine return addresses, and interrupt return addresses. In the system mode, a system

program may set up the USP and activate a routine that runs in that mode, such as when launching an application. In the user mode, all pushes, pops, and subroutine return addresses use the application or user stack. Interrupts, however, always uses the system stack. A user mode program cannot modify the system stack pointer (SSP) and the data segment (DS) register, can only read them. However, both read and write on the extra segment register (ES) is allowed in the user mode. To request use of a different stack or data segment, a user mode program has to call a system mode routine via a TRAP instruction, or signal the system code in some other fashion. A system mode routine can manipulate the segment register (or not, if it decides that the application code shouldn't have access to that area) and return to the application code. In this manner, application code tasks may be easily limited to using certain areas of the total data space. System mode code can use the user stack by copying the user stack pointer (USP) to another pointer register and access the user stack through the data segment register (DS), or the system mode code can use PUSHU and POPU instructions to directly access the user stack. Using these mechanisms, the system monitor can prepare the user stack for a task, such as when launching an application, or easily access parameters on the user stack when it is called by a trap or interrupt instruction for some system service.

Complete programs generally consist of many different modules or segments. However, at any given time during program execution, only a small subset of a program's segments are actually in use. At any given instant, two segments of memory are immediately accessible to an executing program. These are the data segment, where the stack and local variables reside, and the extra segment, which may be used to read remote data structures. Restricting the addressability of a software modules helps gain complete control of system resources for efficient, reliable operation in a multi-tasking environment. A current working data segment address 140 in the microcontroller 12 includes a 16-bit address (pointer) 142 and an 8-bit segment 144 as illustrated in figure 9. The 8-bit segment registers DS or ES holds the offset which is used to identify this current segment. These segment registers are used as extensions to 16-bit pointer registers and stack pointers to allow data to be accessed through the entire 16 megabyte address range. The ES and DS registers are assigned consecutive addresses such that they may be addressed as a single word. There are eight 16-bit registers in the register file. Of those eight, one is reserved for the stack pointer and the other seven may be used as general purpose pointer registers to access the different segments of the memory. A "byte" register in the SFR space contain bits that are associated with each of the seven general purpose pointer registers (i.e not the SP) that selects neither DS or ES

as the source for the most significant 8-bit or for the 24-bit address. This register is called the segment select register or SSEL (see figure 9). The power-on state of the SSEL bits is reset i.e it defaults to the DS register. The stack pointer (SP) does not have a bit in SSEL to determine its segment because it is always segment 0 in system mode and DS in user mode. Segment registers are not automatically incremented or decremented along with their associated pointer registers, but must be altered explicitly by instructions. Writes to the data segment register (manipulating the offset in DS) and writes through the extra segment register (manipulating the memory pointed to by ES:Rn). However, in user mode, writes through DS, writes to ES, and reads through ES are allowed.

Exceptions and interrupts are events that pre-empt normal instruction processing. Exceptions however, unlike interrupts, cannot be masked. Exception and interrupt processing makes the transition from normal instruction execution to execution of a routine that deals with an exception or interrupt. External exceptions are defined as asynchronous, occur as a result of an event external to the processor, and bears no necessary relationship with the current executing program. Examples of asynchronous exceptions in the microcontroller 12 are hardware Reset, and nonmaskable interrupts (NMI). Asynchronous exceptions occur without reference to CPU clocks, but exception processing is synchronized. Internal exceptions, which are defined as synchronous, are caused directly by the currently executing program. The execution of a particular instruction results in the occurrence of such synchronous exceptions, whether intentionally e.g. TRAP, DKPT, software RESET, and TRACE or as an unanticipated exception like divide by 0, write through ES (in not-allowed mode) and stack overflow trap. Exception processing whether synchronous or asynchronous, is always completed, and the first instruction of the handler routine is always executed, before other exceptions are detected. Synchronous and asynchronous exceptions/traps all have the same vector structure like interrupts. Each exception has an assigned vector that points to an associated handler routine. Exception processing includes all operations required to transfer control to a handler routine, but does not include execution of the handler routine itself.

An exception/interrupt vector is the address of a routine that handles an exception or interrupt. Exception/Interrupt vectors are contained in a data structure 150 called the vector table, illustrated in figure 10, which is located in the first 256 bytes of code memory page 0. All vectors include two words which are: (i) the address 152 of the exception/interrupt handler or interrupt and (ii) the initial PSW contents 154 for the handler. The first instruction of each exception/interrupt handler must be in page 0 of program

memory. This is because the address 152 of the handler in the vector table is only 16-bits (page 0). The first instruction of the handler can use a FAR jump or FCALL to switch to a different page of code memory, if needed.

All exceptions and interrupts other than RESET cause the current program counter (PC) and PSW values to be stored on the stack and are serviced after the completion of the current instruction. During an exception or an interrupt, the 24-bit return address 156 and the current PSW word 158 are pushed onto the stack in an arrangement as illustrated in figure 11. The stacked PC (hi-byte): PC (lo-word) value is the 24-bit address of the next instruction in the current instruction stream. The program counter (PC) is then loaded with the address 152 of the corresponding handler routine from the vector table and the PSW is then loaded with a new value stored in the upper word 154 of the corresponding vector.

For interrupts and exceptions, the new PSW holds the interrupt mask (IM) bits for that interrupt reflecting the priority level. Exceptions have a fixed priority that is used to arbitrate between any exceptions that might occur simultaneously. This is called precedence to differentiate from the interrupt priorities associated with the IM bits in the PSW register 44. Precedence is essentially the same mechanism that is used to arbitrate between simultaneous interrupts of the same priority level except that the interrupt priorities are evaluated in the external interrupt controller 84 and the precedence is evaluated by microcode program control in the core 60. Interrupts are not sampled during exception processing, so that the first instruction of a handler is executed before another exception or interrupt can be processed. Execution of the interrupt handler proceeds until the RETI (return-from-interrupt) instruction is encountered or by another exception or an interrupt of higher priority. The RETI instruction terminates each handler routine. Under microcode program control this pops the return address from the stack into the PC, reloads the original PSW from the stack and causes the processor to resume execution of the interrupted routine.

The microcontroller 12 architecture supports up to thirty-five maskable and one non-maskable (NMI) vectored interrupts. The architecture supports two interrupt controller units 84 (outside the core, only one being shown in figure 1 for simplicity) each capable of supporting up to 16 sources. Interrupts from the first unit have a higher precedence than the interrupts from the second interrupt-controller unit. The architecture supports up to sixteen interrupt priority levels for each maskable interrupt. These priorities are stored as a 4-bit value for each interrupt source in interrupt priority (IP) registers 46. There are 4-bits also in the PSW (IMO:3) which reflect the status of the priority level of the interrupt in progress or otherwise elevated code execution priority. The level 0 or No-Priority is represented by the

normal code and 15 indicates the highest priority. The interrupt sources include external hardware interrupts, the timer overflow interrupts, the programmable counter array (PCA) interrupt, UART receiver and transmitter interrupts, the I^2C Interrupt, UPS, and software interrupts. The microcontroller 12 permits all the maskable interrupts to be globally enabled or disabled by means of the global interrupt EA bit in a first interrupt enable (IE) register. Resetting this bit disables all maskable interrupts. Setting it enables all maskable interrupts whose corresponding IE bits are also set. The EA and all maskable interrupt enable bits are initialized to their disable (0) state on RESET. Prioritization of maskable interrupts is performed by the conventional interrupt controller units 84 that exist outside of core 60. These units 84 contain the interrupt enable and priority registers and arbitrate between the interrupt sources connected to them. The arbitration logic conventionally decides which interrupt will be serviced first if two or more interrupts of the same priority level occur simultaneously. The interrupt source with the higher arbitration ranking is serviced first. The interrupt control units 84 also provide the interrupt vector to the execution and decode units 70 and 74 for the maskable interrupts. The external interrupts can be programmed using the bits in the timer control registers 50 to be level or edge sensitive. The enable flags for these external interrupts are bits in the same register. In the event of an external edge sensitive interrupt, the bits are automatically cleared upon vectoring to the corresponding interrupt handler. However, if it is level sensitive, the bit(s) must be reset by the external hardware. The on-chip peripheral interrupts - Timers, UART, etc. are level sensitive; once asserted, they remain so until they are serviced by an appropriate action at the peripheral, or until they are disabled at the peripheral. The timer interrupts are generated by the individual overflow bits in the timer registers 50 in the event of an overflow from the corresponding timer/counter register(s). The timer interrupts generated by these frames are set by the user software but are self-clearing upon vectoring to the appropriate interrupt handling routines. Serial port (UART) interrupts are generated by the transmit and receive interrupt flags (RI and TI). These are enabled by the user software requesting a UART interrupt. These flags are not self-clearing and must be reset in the service routine. The microcontroller 12 supports separate vectors for transmit and receive interrupts. All interrupt enable flags in the microcontroller 12 are user programmable i.e. could be set or reset at any time by the user software overriding the hardware. So interrupts could be generated or pending interrupts can be disabled by the software. However, any write (including bit manipulation) to an interrupt enable register (IE), interrupt priority register (IP), or the PSW will put the current interrupt processing (if any) on hold until the instruction is completed. The microcontroller 12

recognizes 32 interrupt vectors in a table, located at the pr gram memory (EPROM) 76 address 80 Hex. As previously discussed, each vector is composed of two 16-bit words, the first word is the address of the particular interrupt handler. Since this is a one word address, this must be located on code page 0 (0 - 64K). The second word of the vector is an image of the PSW (word) written prior to execution of the handler. This allows changing the priority of the event from the level used by the interrupt controller, and changing the register bank automatically for faster context switching.

A non-maskable interrupt (NMI) can be associated with any interrupt source. In the event an NMI is asserted, the microcontroller interrupt controller 84 simply ignores the interrupt mask bits (IM0:3 in the PSW). Hence, no separate priority level is asserted for NMI. As previously mentioned, the first instruction of all exceptions and interrupt service routines must be located in the page 0 or 16-bit address space (first 64K address space) supported by the XA. This is because the vector table for the microcontroller 12 is located in Page 0 and contains 16-bit addresses only. Also, the handlers for any interrupt or exception that might occur before complete system initialization must reside completely on page 0 since the micrcontroller preferably always initializes itself to page 0 Mode on RESET, e.g. the NMI service routine must be placed in page 0 unless the system guarantees that NMI cannot happen shortly after a reset. The architecture supports up to 8 software interrupts. These are hardware interrupts activated by software rather than by some peripheral action, e.g., timers, etc. Two byte SFRs will contain the flag bits for these software interrupts. These flags are not "self-clearing" and must be reset by the user software. However, these interrupts will be serviced last when at the same priority as a hardware interrupt.

There are several ways in which code or instruction addresses may be formed to execute instructions on the microcontroller 12. Changing the program flow is done with simple relative branches, long relative branches, 24-bit jumps and calls, 16-bit jumps and calls, and returns. Simple relative branches use an 8-bit signed displacement added to the program counter (PC) to generate the new code address. The calculation is accomplished by shifting the 8-bit relative displacement left by one bit (since it is a displacement to a word address), sign extending the result to 24-bits, adding it to the program counter contents, and forcing the least significant bit of the result to zero. The long relative unconditional branch (JMP) and call with 16-bit relative displacements use the same sequence. The branch range is +255 to -256 for 8-bit relative and +65535 to -65536 for long jump and call. Far jumps and calls, which can be used during interrupt processing, include a 24-bit absolute address in the instruction and simply replace the entire program counter contents with the new value. The

address range is anywhere in the 16M address space for the microcontroller 12. Return instructions obtain an address from the stack, which may be either 16 or 24-bits in length, depending on the type of return and the setting of the page zero mode bit in the SCR register. A 24-bit address simply replaces the entire program counter value. A 16-bit return address replaces only the bottom 16 bits of the PC.

The microcode controlled microcontroller processing that occurs to perform the context switch of the present invention starts with an interrupt 200, as illustrated in figure 12. The execution unit 70 (see figure 2) and decode unit 74 both receive the interrupt signal indicating the type or identity of the interrupt from the interrupt controller 84 and the decode unit 74 also receives from the execution unit 70 a signal indicating that an interrupt is occurring. The execution unit 70 continues execution of the currently executing instruction and after execution is finished 202 asserts the interrupt. When asserted the execution unit 70 indicates to the decode unit 74 whether the interrupt is a hardware or software interrupt. The assertion results in the instruction being provided by the fetch unit 78 being blocked and a call interrupt (CALLI) instruction being staged in or forced into the staging register 90 and presented to the execution unit 70. The call interrupt instruction which is staged, because of the particular interrupt signal being provided to the decode unit 74 by the controller 84, includes the address of the interrupt vector for the particular interrupt handler for the particular interrupt. The call interrupt instruction as executed by the execution unit 70 pushes 206 the current program counter (PC), which is pointing at the blocked instruction, onto the system stack (see figure 7) followed by pushing 208 the current program status word (PSW) onto the system stack. The PSW for the interrupt handler found in the interrupt vector is fetched by the fetch unit 78 and loaded 210 into the PSW register 44 by the execution unit 70. This is followed by the fetch and loading 212 of the PC address (also in the vector), of the first instruction of the interrupt handler, into the fetch unit 78. This results in new operating environment parameters being loaded into the microcontroller 12 and a resulting context switch. The fetch unit 78, using the loaded PC, fetches 214 the first instruction of the interrupt handler and normal fetch and execute operations 216 continue until a return-from-interrupt instruction (RETI) is decoded by the decoder 74. At this time the execution unit 70 pops 218 the PSW of the interrupted routine off the system stack and loads 220 it into the PSW register 44. The execution unit then pops 222 the PC of the interrupted routine off the system stack and provides it to the fetch unit 78. The fetch unit 78 then fetches 226 the next instruction (the blocked instruction) of the routine interrupted by the interrupt and normal processing of the interrupted routine continues 228 and another context switch

occurs.

When a system mode program is to launch or initiate an application the system mode program configures and then pushes 250 the PSW of the application to be launched onto the system stack, as depicted in figure 13. The system program then pushes 252 the starting address or PC value of the entry point to the application onto the system stack. Then any other general purpose registers of the register file 82, special function registers 40 or segment registers and user stack that need to be loaded for the start of the application are loaded 254. The system program then executes a return-from-interrupt instruction. This results in the decode 74, execution 70 and fetch 78 units performing the operations 218 - 226 described above with respect to figure 12. In this situation control will return to the system mode whenever an interrupt occurs or exception. The system program must recognize that this interrupt comes from a program launched in this special way and then perform processing accordingly.

The many features and advantages of the invention are apparent from the detailed specification and, thus, it is intended by the appended claims to cover all such features and advantages of the invention which fall within the true spirit and scope of the invention. Further, since numerous modifications and changes will readily occur to those skilled in the art, it is not desired to limit the invention to the exact construction and operation illustrated and described, and accordingly all suitable modifications and equivalents may be resorted to, falling within the scope of the invention.

CLAIMS:

1. A method of context switching between programs during computer program controlled processing, comprising:

a. determining that a context switch from a first program to a second program is to occur;

b. loading a program status word for the second program indicating new operating environment parameters; and

c. starting execution of the second program using the loaded program status word.

2. A method as recited in claim 1, wherein step b. loads one or more f program mode, interrupt priority and register banks.

3. A method as recited in claim 1, further comprising, prior to step c., storing the program status word and the program counter of the first program on a system stack.

4. A method as recited in claim 1, wherein step c. includes loading the program counter of the second program.

5. A method as recited in claim 3, further comprising:

d. determining that a context switch back to the first program from the second program is to occur;

e. retrieving the program status word and program counter of the first program from the system stack;

f. loading the program status word and program counter of the first program; and

g. executing the first program responsive to the loaded program status word anc program counter of the first program.

6. A method as recited in claim 1, wherein the context switch is an interrupt, the second program is an interrupt handler and step a. includes providing an interrupt vector containing the program status word and program counter address of an entry point of the interrupt handler.

7. A method as recited in claim 1, wherein the context switch is an application launch, the second program is the application and step a. includes placing the program status

word and program counter of the application on a system stack, step b. loads from the system stack and step c. comprises performing a return-from-interrupt.

8. A method of context switching between programs during computer program controlled processing, comprising:

a. determining that a context switch from a first program to a second program is to occur;

b. storing a program status word and program context of the first program;

c. loading a program status word for the second program indicating program mode, interrupt priority and register bank set;

d. loading the program counter of the second program;

e. starting execution of the second program using the loaded program status word and the program counter;

f. determining that a context switch back to the first program from the second program is to occur;

g. retrieving the program status word and program counter of the first program from the system stack;

h. loading the program status word and program counter of the first program; and

i. executing the first program responsive to the loaded program status word and program counter of the first program.

9. An apparatus for performing a context switch between programs during computer program controlled processing, comprising:

a computer, including:

means for loading a program status word of a program to be executed where the program status word indicates a mode of the program, registers used by the program and interrupt priority of the program; and

means for executing the program responsive to the program status word.

10. A program controlled apparatus comprising:

a. means for determining that a context switch from a first program to a second program is to occur;

b. means for loading a program status word for the second program indicating new operating environment parameters; and

c. means for starting execution of the second program using the loaded program status word.

11. An apparatus as recited in claim 10 wherein the means for loading are adapted to load one or more of program mode, interrupt priority and register banks.

12. An apparatus as recited in claim 10, further comprising means for storing, prior to the start of execution of the second program, the program status word and the program counter of the first program on a sysem stack.

13. An apparatus as recited in claim 10, wherein the storing means are adapted to load the program counter of the second program.

14. An apparatus as recited in claim 12, further comprising:

d. means for determining that a context switch back to the first program from the second program is to occur;

e. means for retrieving the program status word and program counter of the first program from the system stack;

f. means for loading the program status word and program counter of the first program; and

g. means for executing the first program responsive to the loaded program status word and program counter of the first program.

15. An apparatus as recited in claim 10, wherein the context switch is an interrupt, the second program is an interrupt handler and the determining means are adapted to provide an interrupt vector containing the program status word and program counter address of an entry point of the interrupt handler.

16. An apparatus as recited in claim 10, wherein the context switch is an application launch, the second program is the application and the determining means are adapted to place the program status word and program counter of the application on a system stack, and the loading means are adapted to load from the system stack and the starting means comprises means for performing a return-from-interrupt.

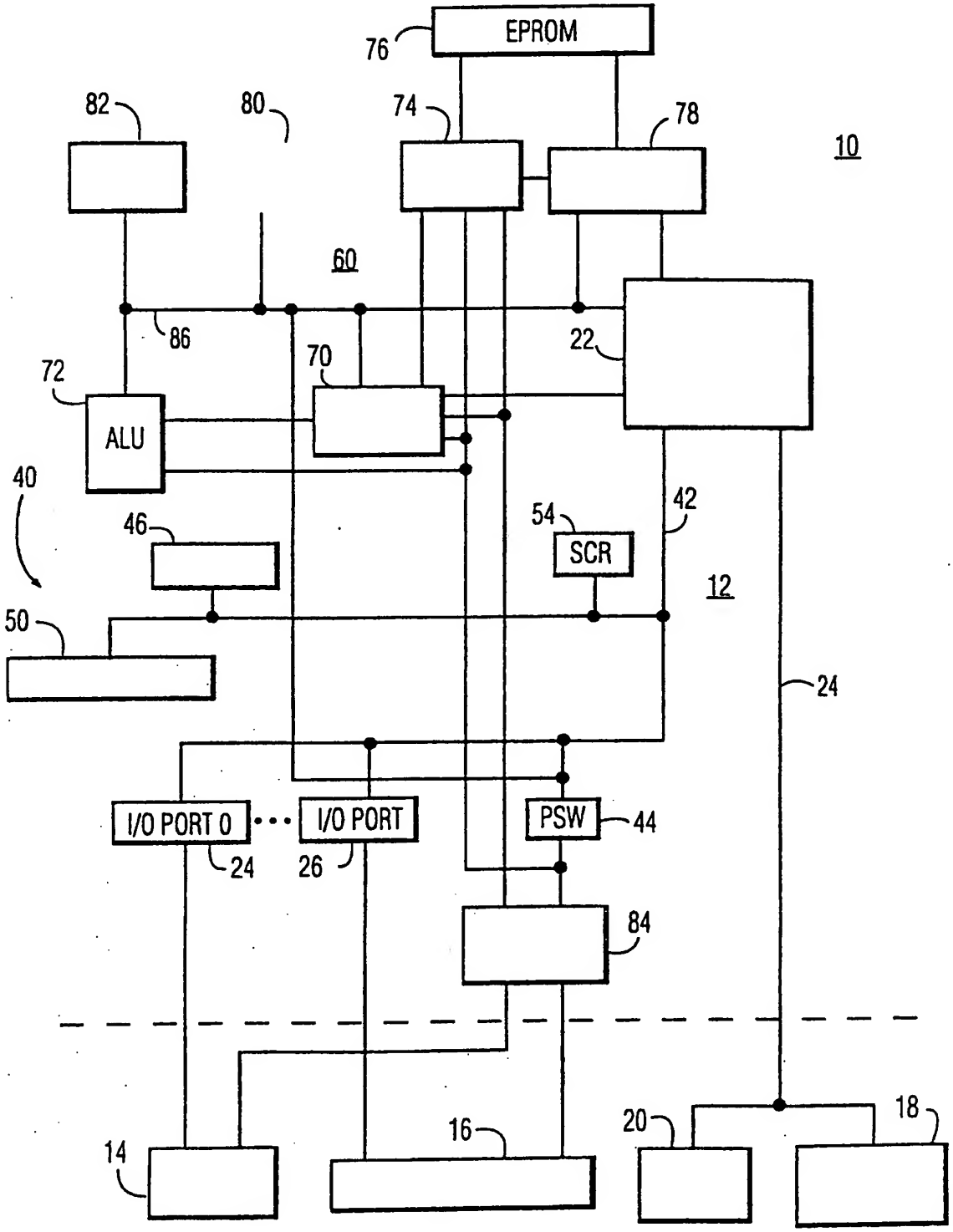

EPROM
76
82
80
74
78
10
60
86
22
72
70
ALU
40
46
54
SCR
42
12
50
24
I/O PORT 0
I/O PORT
PSW
44
24
26
84
14
16
20
18

FIG. 1

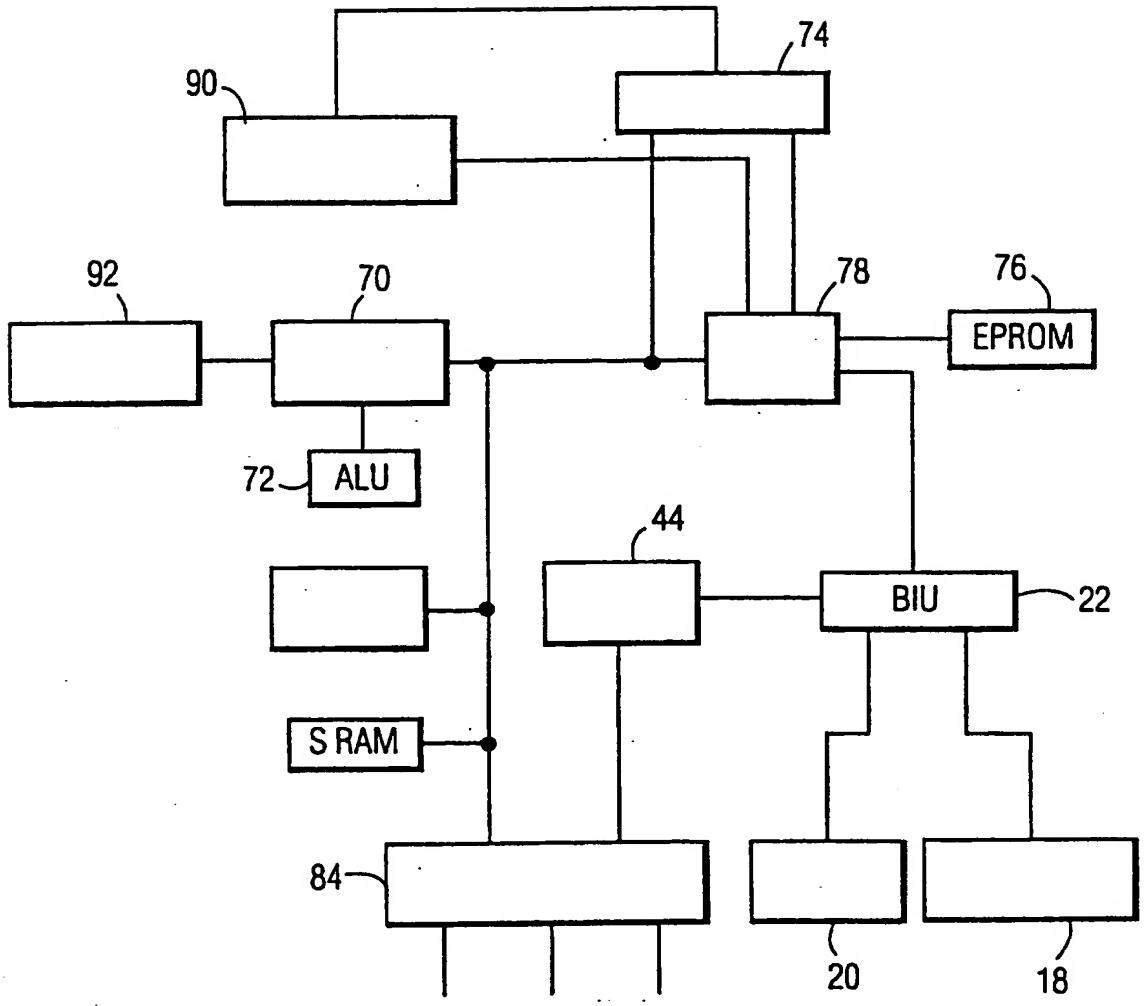


FIG. 2

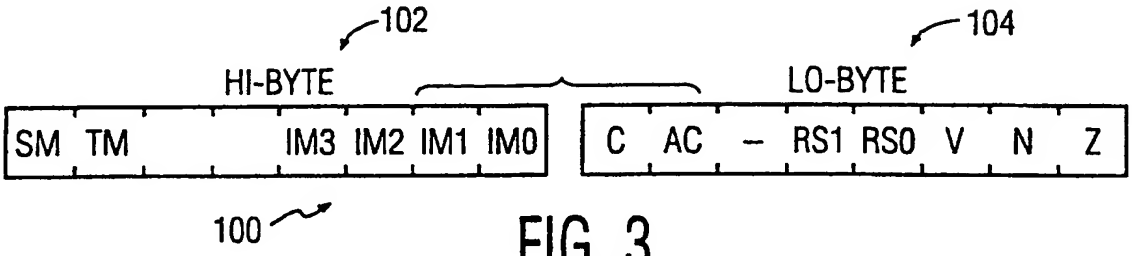

102
104
HI-BYTE
LO-BYTE
SM TM IM3 IM2 IM1 IM0
C AC – RS1 RS0 V N Z
100

FIG. 3

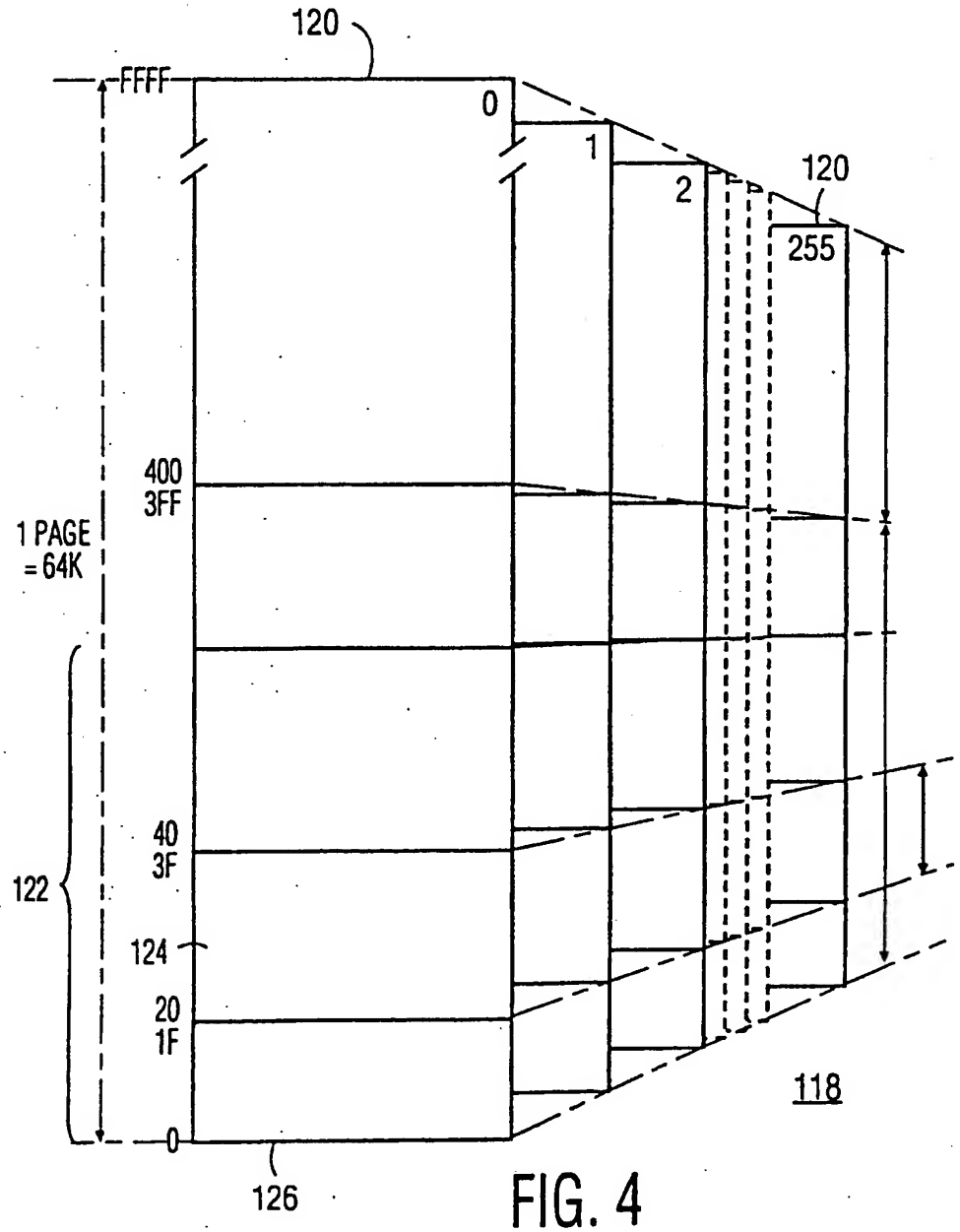

120
FFFF
0
1
2
120
255
400
3FF
1 PAGE
= 64K
40
3F
122
124
20
1F
0
118
126

FIG. 4

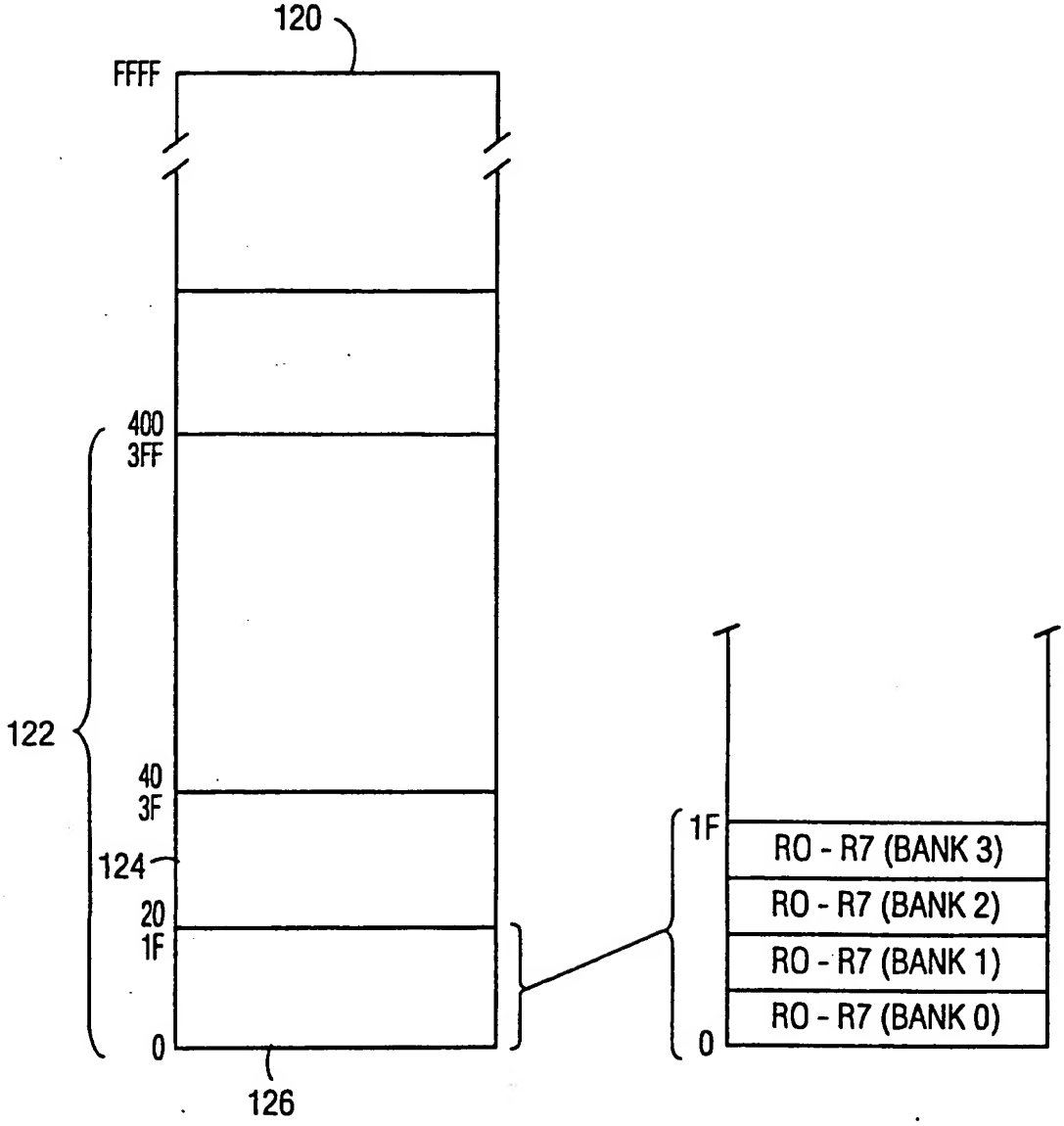

120
FFFF
400
3FF
122
40
3F
124
20
1F
0
126
1F
R0 - R7 (BANK 3)
R0 - R7 (BANK 2)
R0 - R7 (BANK 1)
R0 - R7 (BANK 0)
0

FIG. 5

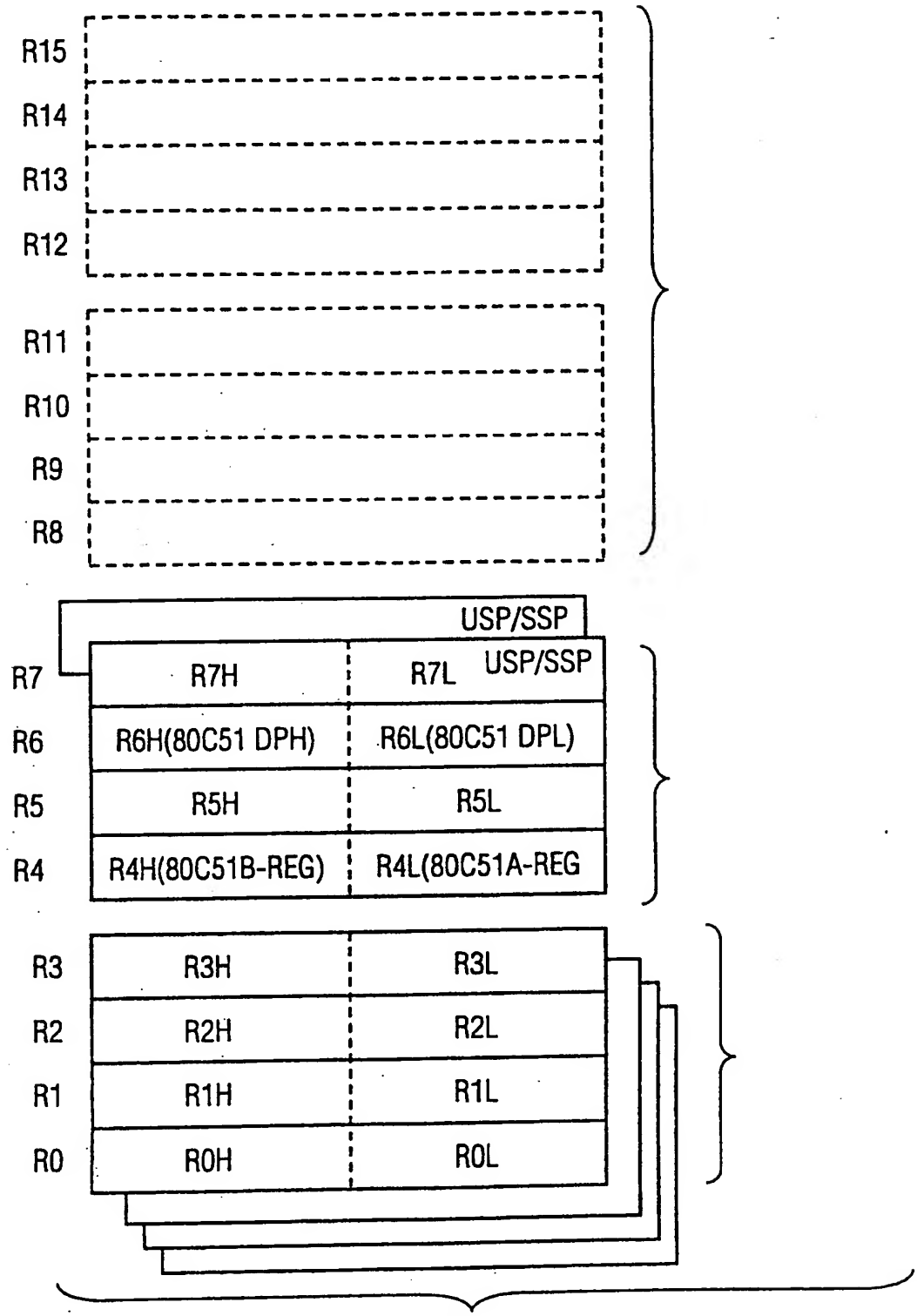

R15
R14
R13
R12
R11
R10
R9
R8
USP/SSP
R7
R7H
R7L
USP/SSP
R6
R6H(80C51 DPH)
R6L(80C51 DPL)
R5
R5H
R5L
R4
R4H(80C51B-REG)
R4L(80C51A-REG
R3
R3H
R3L
R2
R2H
R2L
R1
R1H
R1L
R0
R0H
R0L

FIG. 6

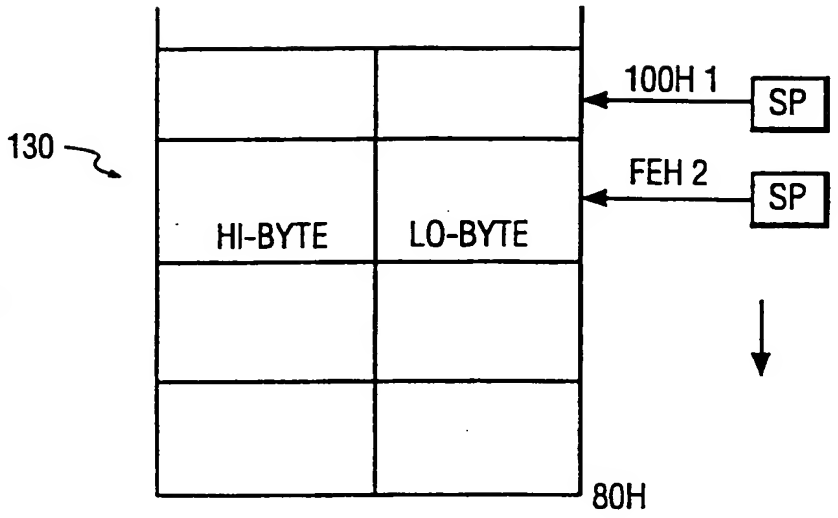


FIG. 7

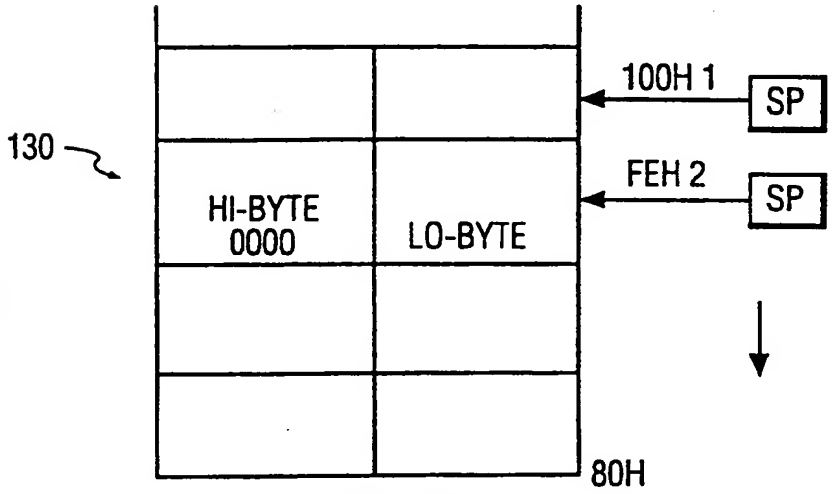


FIG. 8

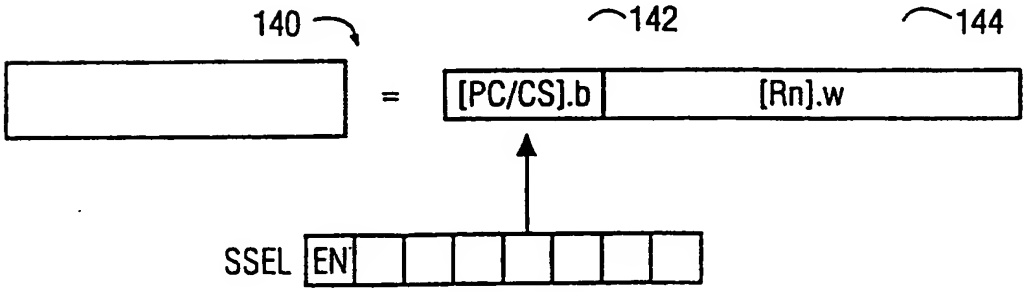


FIG. 9

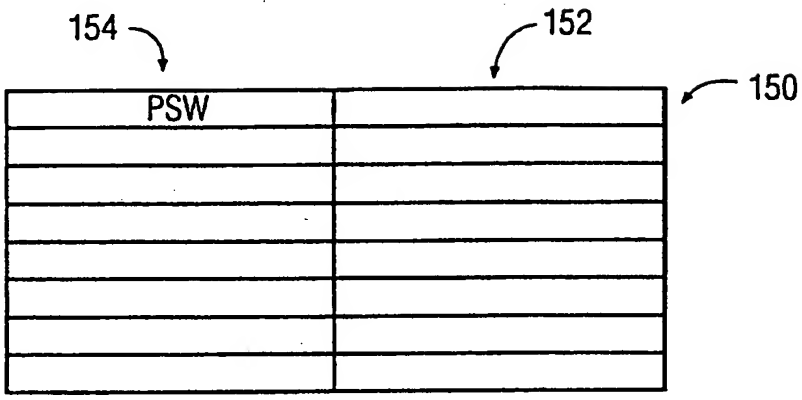


FIG. 10

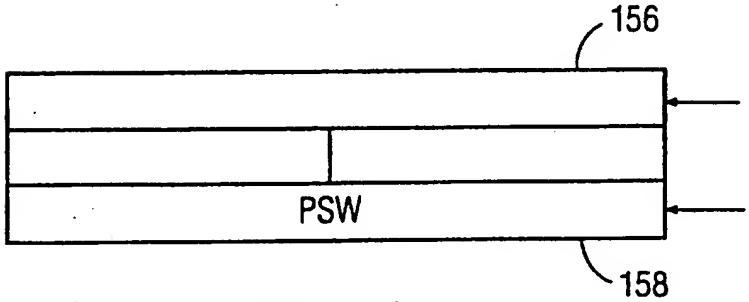


FIG. 11

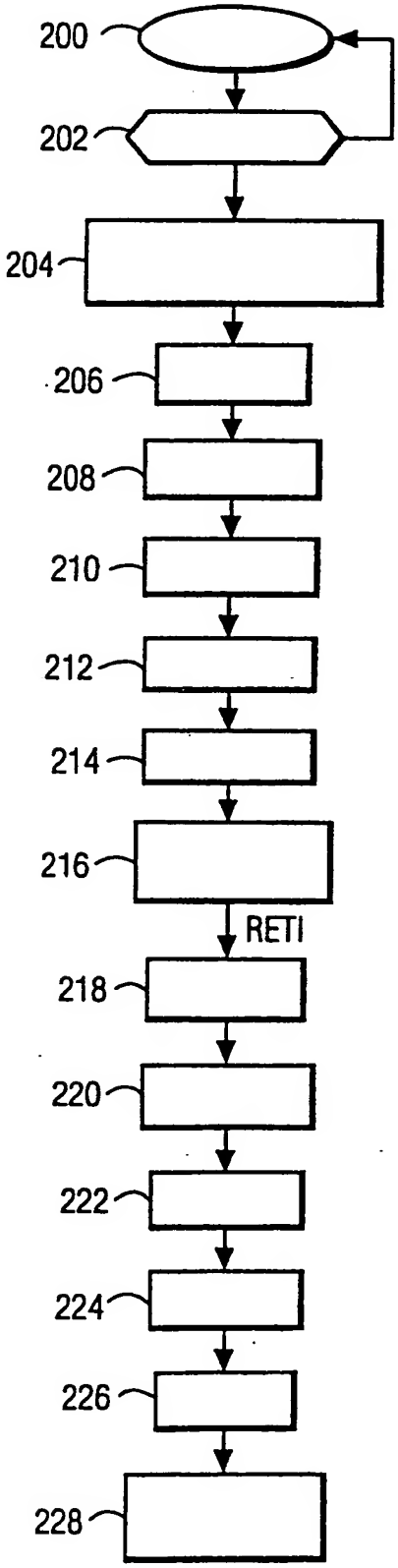


FIG. 12

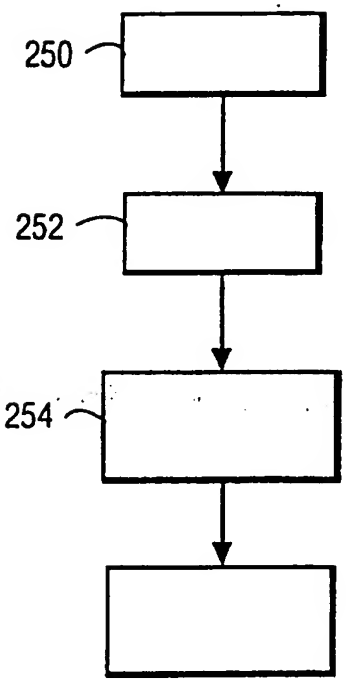


FIG. 13

THIS PAGE BLANK (USPTO)